Introduction to Algorithms September 28, 2001
Massachusetts Institute of Technology 6.046J/18.410J
Singapore-MIT Alliance SMA5503
Professors Erik Demaine, Lee Wee Sun, and Charles E. Leiserson Handout 12

Problem Set 2 Solutions

MIT students: This problem set is due in lecture on *Monday, September 24*.

SMA students: This problem set is due after the video-conferencing session on *Wednesday, September 26*.

Reading: Chapters 6, 7, §5.1-5.3.

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered by the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation instructor and time, the date, and the names of any students with whom you collaborated.

MIT students: Each problem should be done on a separate sheet (or sheets) of three-hole punched paper.

SMA students: Each problem should be done on a separate sheet (or sheets) of two-hole punched paper.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions.

Exercise 2-1. Do Exercise 5.3-1 on page 104 of CLRS.

Solution:

RANDOMIZE-IN-PLACE
$n \leftarrow length[A]$
Exchange $A[1] \leftrightarrow A[Random(1, n)]$
for $i \leftarrow 2$ to n
do Exchange $A[i] \leftrightarrow A[Random[i, n]$

For our base case, we have i initialized to 2. Therefore we must show that for each possible 1-permutation, the subarray A[1] contains this 1-permutation with probability $(n-i+1)!/n! = 1/n$. Clearly this is the case, as each element has a chance of $1/n$ of being in the first position.

Exercise 2-2. Do Exercise 6.1-2 on page 129 of CLRS.

Solution:

By definition, a 1 element heap has a height of 0. Therefore $\lfloor \lg n \rfloor$ where $n = 1$ is 0 and our base case is correct.

Now we use induction, and assume that all trees with $n - 1$ nodes or fewer has a height of $\lfloor \lg n \rfloor$. Next we consider a tree with n nodes. Looking at the n node, we know its height is one greater than its parent (and since we're not in the base case, all nodes have a parent). The parent of the nth node in the tree is also the $\lfloor n/2 \rfloor$th node in the tree. Therefore its height is $\lfloor \lg \lfloor n/2 \rfloor \rfloor$. Then the nth node in the tree has a height of $1 + \lfloor \lg \lfloor n/2 \rfloor \rfloor = \lfloor 1 + \lg \lfloor n/2 \rfloor \rfloor = \lfloor \lg 2 + \lg \lfloor n/2 \rfloor \rfloor = \lfloor \lg n \rfloor$. Therefore by induction we have shown that the height of an n node tree is $\lfloor \lg n \rfloor$.

Exercise 2-3. Do Exercise 6.4-3 on page 136 of CLRS.

Solution:

The running time of HEAPSORT on an array A of length n that is already sorted in increasing order is $\Theta(n \lg n)$ because even though it is already sorted, it will be transformed back into a heap and sorted.

The running time of HEAPSORT on an array A of length n that is sorted in decreasing order will be $\Theta(n \lg n)$. This occurs because even though the heap will be built in linear time, every time the max element is removed and the HEAPIFY is called it will cover the full height of the tree.

Exercise 2-4. Do Exercise 7.2-2 on page 153 of CLRS.

Solution:

The running time will be $\Theta(n^2)$ because every time partition is called, all of the elements will be put into the subarray of elements smaller than the partition. The recurrence will be $T(n) = T(n-1) + \Theta(n)$ which is clearly $\Theta(n^2)$

Exercise 2-5. Do Problem 7-3 on page 161 of CLRS.

Solution:

(a) This sort is intuitively correct because the largest $1/3$rd of the elements will eventually be sorted amung their peers. If they are in the first third of the array to begin with, they will be sorted into the middle third. If they are in the middle or last third, then they will obviously be sorted into their proper position. Similarly any element which belongs in the each of the thirds will be sorted into position by the three sub-sorts.

(b)

$$T(n) = 3T(2n/3) + \Theta(1)$$

Which solves to $\Theta(n^{\log_{1.5} 3}) \approx n^{2.7}$

(c) STOOGESORT is slower than all of the other algorithms we have studied. INSERTION $= \Theta(n^2)$, MERGE SORT $= \Theta(n \lg n)$, HEAPSORT $= \Theta(n \lg n)$, and QUICKSORT $= n^2$. Therefore all other sorts are faster and these professors do not deserve tenure for this work!

Problem 2-1. Average-case performance of quicksort

We have shown that the expected time of randomized quicksort is $O(n \lg n)$, but we have not yet analyzed the average-case performance of ordinary quicksort. We shall prove that, under the assumption that all input permutations are equally likely, not only is the running time of ordinary quicksort $O(n \lg n)$, but it performs essentially the same comparisons and exchanges between input elements as randomized quicksort.

Consider the implementation of PARTITION given in lecture on a subarray $A[p\,..\,r]$:

```
PARTITION(A, p, r)
 1  x ← A[p]
 2  i ← p
 3  for j ← p + 1 to r
 4       do  if A[j] ≤ x
 5             then i ← i + 1
 6                  exchange A[i] ↔ A[j]
 7  exchange A[p] ↔ A[i]
 8  return i
```

Let S be a set of distinct elements which are provided in random order (all orders equally likely) as the input array $A[p\,..\,r]$ to PARTITION, where $n = r - p + 1$ is the size of the array. Let x denote the initial value of $A[p]$.

(a) Argue that $A[p+1\,..\,r]$ is a random permutation of $S - \{x\}$, that is, that all permutations of the input subarray $A[p+1\,..\,r]$ are equally likely.

Solution:

Given that $A[p\,..\,r]$ is random (all orders are equally likely), there are $n!$ possible permutations of the $n = r - p + 1$ elements. Each element has a $1/n$ probability of being chosen as the pivot, therefore the number of permutations of the remaining elements is $n! \cdot 1/n = (n-1)!$. Consequently the $(n-1)!$ permutations are equally likely.

Define $\delta : S \to \{-1, 0, +1\}$ as follows:

$$\delta(s) = \begin{cases} -1 & \text{if } s < x\,, \\ 0 & \text{if } s = x\,, \\ +1 & \text{if } s > x\,. \end{cases}$$

(b) Consider two input arrays $A_1[p\,..\,r]$ and $A_2[p\,..\,r]$ consisting of the elements of S such that $\delta(A_1[i]) = \delta(A_2[i])$ for all $i = p, p+1, \ldots, r$. Suppose that we run PARTITION on $A_1[p\,..\,r]$ and $A_2[p\,..\,r]$ and trace the two executions to record the branches taken, indices calculated, and exchanges performed — but not the actual array values manipulated. Argue briefly that the two execution traces are identical. Argue further that PARTITION performs the same permutation on both inputs.

Solution:

PARTITION takes different branches based only on the comparisons made in the δ function. This is clear by observing line 4 of the function, as it is the only place where the sequence of instructions may differ. As the arrays have identical $\delta()$ function values, they must take the same branches, calculate the same indicies and perform the same exchanges. Consequently PARTITION will perform the same partition on both arrays, which follows directly as the exchanges performed are identical.

Define a sequence $F = \langle f_1, f_2, \ldots, f_n \rangle$ to be an (n, k) ***input pattern*** if $f_1 = 0$, $f_i \in \{-1, +1\}$ for $i = 2, 3, \ldots, n$, and $|\{i : f_i = -1\}| = k - 1$.

Define a sequence $F = \langle f_1, f_2, \ldots, f_n \rangle$ to be an (n, k) ***output pattern*** if

$$f_i = \begin{cases} -1 & \text{if } i < k\,, \\ 0 & \text{if } i = k\,, \\ +1 & \text{if } i > k\,. \end{cases}$$

We say that a permutation $\langle s_1, s_2, \ldots, s_n \rangle$ of S ***satisfies*** a pattern $F = \langle f_1, f_2, \ldots, f_n \rangle$ if $\delta(s_i) = f_i$ for all $i = 1, 2, \ldots, n$.

(c) How many (n, k) input patterns are there? How many (n, k) output patterns are there?

Solution:

There are $\binom{n-1}{k-1}$ (n, k) input patterns because we can choose $k-1$ positions out of $n-1$ possible positions to have $\delta = -1$. There is one (n, k) output pattern because the pattern must be exactly $k-1$ negative ones followed by a 0, followed by $n-k$ ones.

(d) How many permutations of S satisfy a particular (n, k) input pattern? How many permutations of S satisfy a particular (n, k) output pattern?

Solution:

$(n-k)!(k-1)!$ permutations are possible of S to satisfy a particular input pattern. This is the total number of ways to rearrange the elements which have a δ value of -1 amongst themselves, and rearrange those with a value of 1 amongst themselves. There are also $(n-k)!(k-1)!$ permutations possible to satisfy a particular output pattern for the same reason.

Let $F = \langle f_1, f_2, \ldots, f_n \rangle$ be an (n, k) input pattern, and let $F' = \langle f'_1, f'_2, \ldots, f'_n \rangle$ be an (n, k) output pattern. Define $S|_F$ to be the set of permutations of S that satisfy F, and likewise define $S|_{F'}$ to be the set of permutations of S that satisfy F'.

(e) Argue that PARTITION implements a bijection from $S|_F$ to $S|_{F'}$. (*Hint:* Use the fact from group theory that composing a fixed permutation with each of the $n!$ possible permutations yields the set of all $n!$ permutations.)

Solution:

All members of $S|_F$ satisfy F and so they all have the same result when the δ function is applied to its elements. Therefore by part (b) when all these inputs are given to PARTITION they are subject to the same permutation. Using the hint, we then know that after all of the distinct inputs are run through PARTITION that they will produce all $(n-k)!(k-1)!$ distinct outputs. From part (d) we know that $S|_F$ and $S|_{F'}$ are the same size, and also we have proven that PARTITION is onto, and therefore PARTITION must be a bijection!

(f) Suppose that before the call to PARTITION, the input subarray $A[p+1\,..\,r]$ is a random permutation of $S - \{x\}$, where $x = A[p]$. Argue that after PARTITION, the two resulting subarrays are random permutations of their respective elements.

Solution:

Using our solution from part (e), we know that after PARTITION is run on $S|_F$, we get all values in the set $S|_{F'}$. Therefore we get all permutations of the $n-k$ ones and all permutations of the $k-1$ negative ones. Furthermore, we get each sub-array permutations an equal number of times and so the subarrays are also random permutations.

(g) Use induction to show that, under the assumption that all input permutations are equally likely, at each recursive call of QUICKSORT(A, p, r), every element of S belonging to $A[p\,..\,r]$ is equally likely to be the pivot $x = A[p]$.

Solution:

The base case for the initial array; we know that it is randomly permuted, and so by part (f) and (a) each of its subarrays will also be randomly permuted after PARTITION. Therefore we can inductively apply (f) at each partition to prove that every subarray will also be randomly permuted.

(h) Use the analysis of RANDOMIZED-QUICKSORT to conclude that the average-case running time of QUICKSORT on n elements is $O(n \lg n)$.

Solution:

By part (g) we know that under the assumption that the input pattern is random every element is equally likely to be chosen as the pivot at each recursive call which then produces the same random distribution of quicksort traces as RANDOMIZED-QUICKSORT. Therefore as their distribution is the same, the expected-case analysis for RANDOMIZED-QUICKSORT will apply to the average case of QUICKSORT. Therefore the average case of QUICKSORT also takes $O(n \log n)$ time.

Problem 2-2. Analysis of d-ary heaps

A d-***ary heap*** is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of 2 children.

(a) How would you represent a d-ary heap in an array?

Solution:

The d-ary heap would be similar to a binary heap with the parent and child indexes calculated as follows:
$Parent[i] = \lfloor i/d \rfloor$
jth-Child[i] $= d \cdot i + j$ where $j = 0 \ldots d - 1$
The root of the tree would be at index $i = 1$.

Alternate Solution: A d-ary heap can be represented in a 1-dimensional array as follows. The root is kept in $A[1]$, its d children are kept in order in $A[2]$ through $A[d+1]$, their children are kept in order in $A[d+2]$ through $A[d^2+d+1]$, and so on. The two procedures that map a node with index i to its parent and to its jth child (for $1 \le j \le d$), respectively, are;

D-ARY-PARENT(i)
  return $\lceil (i-1)/d \rceil$

D-ARY-CHILD(i, j)
  return $d(i-1) + j + 1$

To convince yourself that these procedures really work, verify that

$$\text{D-ARY-PARENT}(\text{D-ARY-CHILD}(i, j)) = i \,,$$

for any $1 \leq j \leq d$. Notice that the binary heap procedures are a special case of the above procedures when $d = 2$.

(b) What is the height of a d-ary heap of n elements in terms of n and d?

Solution:

CORRECTION
A d-ary heap would have a height of $\Theta(\log_d n)$. We know that

$$\begin{aligned} 1 + d + d^2 + \ldots + d^{h-1} \;&<\; n \;\leq\; 1 + d + d^2 + \ldots + d^h \\ \frac{d^h - 1}{d-1} \;&<\; n \;\leq\; \frac{d^{h+1} - 1}{d-1} \\ d^h \;&<\; n(d-1) + 1 \;\leq\; d^{h+1} \\ h \;&<\; \log_d(n(d-1)+1) \;\leq\; h + 1 \end{aligned}$$

which solves to $h = \lceil (\log_d(n(d-1)+1) - 1) \rceil$.

(c) Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

Solution:

HEAPIFY(A, i, n, d)
1 $j \rightarrow i$
2 **for** $k \leftarrow 0$ to $d-1$
3 **if** $d * i + k \leq n$ and $A[d * i + k] > A[j]$
4 **then** $j = d * i + k$
5 **if** $j \neq i$
6 **then**
7 Exchange $A[i] \leftrightarrow A[j]$
8 HEAPIFY(A, j, n, d)

The running time of HEAPIFY is $O(d \log_d n)$ because at each depth we are doing d loops, and we recurse to the depth of the tree. In HEAPIFY we compare the the ith node and each of its children to find the maximum value for all of the nodes. Then if

the maximum child is greater than the ith node, we switch the two nodes and recurse on the child.

EXTRACT-MAX(A, n)
1 $max \leftarrow A[1]$
2 $A[1] \leftarrow A[n]$
3 $n = n - 1$
4 HEAPIFY(A, 1, n, d)
5 return max

The running time of this algorithm, is clearly constant work plus the time of HEAPIFY which as shown above is $O(d * log_d n)$. EXTRACT-MAX works by storing the value of the maximum element, moving the minimum element into the root of the heap, and then calling heapify to restore the heap property.

(d) Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

Solution:

See next problem part for INCREASE-KEY definition.

INSERT(A, k, n, d)
1 $n \leftarrow n + 1$
2 $A[n] = -\infty$
3 INCREASE-KEY(A, i, k, n)

From the following problem part, we know INCREASE-KEY runs in $O(\log_d n)$ time, therefore since INSERT only adds constant time operations it is also $O(\log_d n)$. It is rather trivially correct as the algorithm has not changed because all calculations involving the number of children are performed by INCREASE KEY.

(e) Give an efficient implementation of INCREASE-KEY(A, i, k), which first sets $A[i] \leftarrow \max(A[i], k)$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

Solution:

```
INCREASE-KEY(A, i, k)
1  A[i] ← max(A[i], k)
2  if k = A[i]
3     while i > 1 and A[⌊i/d⌋] < A[i]
4        do
5           Exchange A[i] ↔ A[⌊i/d⌋]
6           i ← ⌊i/d⌋
```

Our implementation loops proportionally to at most the depth of the tree, therefore it runs in $O(log_d n)$ time. INCREASE-KEY loops, at each step comparing the increased node to its parent and exchanging them if the heap property is violated. Therefore, once the algorithm terminates we know that the node once again satisfies the heap property and has the correct value.

(f) When might it be better to use a d-ary heap instead of a binary heap?

Solution:

It would be better to use a d-ary heap when it is predicted that the heap will do many more INSERTs and INCREASE-KEYs than EXTRACT-MAXs because INSERT and INCREASE-KEY are faster algorithms as d increases while EXTRACT-MAX gets slower.